Objective

- To master Reduction Trees, arguably the most widely used parallel computation pattern
  - Basic concept
  - A class of computation
  - Memory coalescing
  - Control divergence
  - Thread utilization

Partition and Summarize

- A commonly used strategy for processing large input data sets
  - There is no required order of processing elements in a data set (associative and commutative)
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- We will focus on the reduction tree step for now.
- Google and Hadoop MapReduce frameworks are examples of this pattern

Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations
  - Privatization
    - Multiple threads write into an output location
    - Replicate the output location so that each thread has a private output location
    - Use a reduction tree to combine the values of private locations into the original output location
What is a reduction computation

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
  - Often with user defined reduction operation function as long as the operation
    - Is associative and commutative
    - Has a well-defined identity value (e.g., 0 for sum)
- An important category of “collective” operations in parallel computing.

An efficient sequential reduction algorithm performs N operations - O(N)

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- Iterate through the input and perform the reduction operation between the result value and the current input value

A parallel max reduction tree algorithm performs N-1 Operations in log(N) steps

A tournament is a reduction tree with “max” operation

A more artful rendition of the reduction tree.
A Quick Analysis

- For N input values, the reduction tree performs:
  \((1/2)N + (1/4)N + (1/8)N + \cdots (1/N)N = (1- (1/N))N = N-1\) operations.
- In \(\log(N)\) steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
  - Average Parallelism \((N-1)/\log(N)\)
    - For \(N = 1,000,000\), average parallelism is 50,000.
    - However, peak resource requirement is 500,000!
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to sequential
  - Many parallel algorithms are not work efficient

A Sum Reduction Example

- Parallel implementation:
  - Recursively halve the # of threads, add two values per thread in each step
  - Takes \(\log(n)\) steps for \(n\) elements, requires \(n/2\) threads
- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0
  - Reduces global memory traffic due to partial sum values

Vector Reduction Thread to Data Mapping with Branch Divergence

Simple Thread Index to Data Mapping

- Each thread is responsible of an even-index location of the partial sum vector
  - One input value is at the location of responsibility
- After each step, half of the threads are no longer needed
- In each step, one of the inputs comes from an increasing distance away
A Sum Example

Data

Thread 0 Thread 1 Thread 2 Thread 3
3 1 7 0 4 1 6 3

Active Partial Sum elements

The Reduction Steps

```c
// Stride is distance to the next value being accumulated into the threads mapped position in the partialSum[] array
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t] += partialSum[2*t+stride];
}
```

Why do we need __syncthreads()?

Barrier Synchronization

• __syncthreads() are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

• Why do we not need another __syncthreads() at the end of the reduction loop?

Example of the Purpose of Syncthreads

Data

Thread 0 Thread 1 Thread 2 Thread 3
3 1 7 0 4 1 6 3

Active Partial Sum elements
“Segmented Reduction”

A list of arbitrary length

Global Memory

PartialSum Block 0
PartialSum Block 1
PartialSum Block 2
...
PartialSum Block n-1

Global Memory

Copy back to host and host to finish the total

Back to the Global Picture

- At the end of the kernel execution, thread 0 in each block writes the sum of the block in partialSum[0] into a vector indexed by the value of blockIdx.x
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
  - Could use atomic operations to accumulate into a global sum (to be covered soon).
- If there are only a small number of sums, the host can simply transfer the data back and add them together.

Some Observations

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources
- No more than half of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if-condition, poor resource utilization but no divergence.
    - This can go on for a while, up to 5 more steps (1024/32=32= 2^5), where each active warp only has one productive thread until all warps in a block retire
  - Some warps will still succeed, but with divergence since only one thread will succeed

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- Reduction satisfies this criterion.
A Better Strategy

- Always compact the partial sums into the first locations in the partialSum[] array
- Keep the active threads consecutive

A Better Reduction Kernel

```c
for (unsigned int stride = blockDim.x;
    stride >= 1;  stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

A Quick Analysis

- For a 1024 thread block
  - Each block loads 2048 elements into shared memory
  - No divergence in the first 5 steps
  - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step, threads in each war either all active or all inactive
  - The final 5 steps will still have divergence
A Story about an Old Engineer

- If time permits.

Parallel Algorithm Overhead

```c
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x; stride >= 1; stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

Parallel Execution Overhead

Although the number of “operations” is N, each “operation involves much more complex address calculation and intermediate result manipulation.

If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm.
ANY MORE QUESTIONS?